# Model Consistency for Distributed Collaborative Modeling

Gerson Gerson Sunyé

# Model Consistency for Distributed Collaborative Modeling

Gerson Sunyé
gerson.sunye@univ-nantes.fr
LS2N – University of Nantes
AtlanMod Group (Inria, IMT Atlantique, and LS2N)

November 7, 2017

**Abstract**

Current collaborative modeling tools use a centralized architecture, based on version control system, where models are updated asynchronously. These tools depend on a single server and are not completely adapted for collaborative modeling, where update reactivity is essential. In this paper, we propose a framework for building collaborative modeling tools which provides synchronous model update. The framework is based on a peer-to-peer architecture and uses a consistency algorithm for model updating.

## 1 Introduction

As collaborative modeling becomes more and more popular, changing the way that modelers interact with colleagues to design and create documents, there is a growing need for tools and techniques that enable effective collaboration. A first response for this need is the emergence of online web-based modeling tools, e. g., Lucidchart [24] or GenMyModel [9], and of standalone modeling tools coupled with control version systems, as the recent release of MetaEdit+ [31].

In this paper, we propose a model consistency approach for providing the bases of collaborative modeling tools. This approach is inspired from cooperative editing systems, introduced in Section 2.1 and is based on the Eclipse Modeling Framework [26], EMF, the *de-facto* standard framework for building modeling tools, which is introduced in Section 2.2.

The goal of our approach is to provide the basis for developing modeling tools with following characteristics: (i) *distributed*: collaborative tools can be deployed on distributed nodes, connected by networks with different latency times, and do not require a centralized server for update integration; (ii) *reactive*: the response for integrating remote updates is fast with low latency; (iii) *synchronous*: local updates are broadcast to other nodes right after their execution.

Differently from other approaches that use a generic control-version server, e. g., Git or SVN, or a model-specific one, e. g., EMFStore [12], the approach does not use versions and resolves conflicts automatically aiming at a simple goal, that all model replicas are consistent. The advantages and limits of the approach with respect to other research efforts are discussed in Section 5.

To ensure that remote changes are integrated with the same execution order in all nodes, the approach classifies the relations between updates into four distinct types: independent, dependent, equivalent, and conflictual. Independent updates can be executed in any order, while dependent ones must always follow the same order. Equivalent updates produce the same result and thus only one should be executed, and conflictual updates produce different results depending on their execution order. The integration of the latter is more complex and may result in undoing local changes and re-executing them after the integration. Section 3 describes the approach an the integration algorithm, as well as a simple example that illustrates the approach.

To validate the integration algorithm through implementation, we develop a prototype that uses EMF notifications to capture local updates and the publish–subscribe architectural pattern [4] to broadcast them to remote nodes. Section 4 describes the implementation.

# 2 Background

This section introduces the principles of cooperative editing systems, which inspired our work, and some of the modeling concepts implemented in EMF that help the comprehension of the model consistency approach.

## 2.1 Cooperative Editing Systems

A real-time cooperative editing system consists of a set of interconnected nodes where locally to each node, users perform changes on a shared document. Each node propagates its local changes to the remote nodes, which integrate them to the local copy of the shared document. The system maintains the consistency among the different copies. A cooperative editing system is said to be consistent if it maintains the following properties [29]:

**Convergence.** When the same set of operations have been executed at all nodes, all copies of the shared document are identical.

**Causality Preservation.** For any pair of operations $O_a$ and $O_b$ , if $O_a \rightarrow O_b$, then $O_a$ is executed before $O_b$ at all nodes.

**Intention Preservation.** For any operation $O$, the effects of executing $O$ at all sites are the same as the intention of $O$, and the effect of executing $O$ does not change the effects of independent operations.

A common solution to achieve consistency is to use an operational transformation approach [28], which consists of an integration algorithm and a transformation function. The integration algorithm is responsible for performing, broadcasting, and receiving operations, while the transformation function is responsible for detecting and merging concurrent operations. The transformation function often relies on vector clocks, e. g., GOTO [28] and ABT [18]. A vector clock is an array of logical clocks, one clock per node, associated to each operation and used to determine the causality between operations. The limit of vector clocks is that the size of the exchanged messages grows with the number of nodes, creating a bottleneck that prevents these systems to scale. A scalable alternative to vector clocks is to use semantic causal dependency [16, 20], declared with respect to operation preconditions. For instance, consider a Graph on which two operations are performed, $O_1 = createVertex(A)$ and $O_2 = createVertex(B)$. There is no casual dependency between this two operations since their execution order can be interchanged. However, if a third operation $O_3 = createEdge(A, B)$ is considered, then there is a casual dependency: the execution of $O_3$ requires that vertices A and B exist, i. e., $O_3$ must be executed after $O_1$ and $O_2$.

While cooperative editing systems focus on documents and on casual dependency of operations on characters, we believe that their techniques and algorithms can also be applied to structured data models.

## 2.2 EMF

The Eclipse Modeling Framework is a set of components that aims at helping developers to create sophisticated modeling tools [26]. Similarly to other modeling frameworks, e. g., MDR [19] and NSUML [22], it proposes a modeling language, Ecore, and code generation facilities to create Java underlying models, specific to each Ecore model. In EMF terms, the Java generated modeling elements are (subclasses of) **EObject** and their meta-types, the elements of an Ecore model, are instances of **EClass**. Unlike the other frameworks, EMF introduces the concept of *Resource*, a container for modeling element instances (**EObject** sub-instances), which is independent from Ecore models. Indeed, a resource can contain a subset of instances from the same underlying model, as well as instances from other models. Resources are mainly used to persist instances on different formats: e. g., XMI, relational databases [25], or NoSQL databases [21, 7].

Resources respect the containment relationship: when an instance is attached to a resource, so are all its contents. Conversely, when an instance is detached from a resource, all its contents are also detached. Resources are responsible for assigning identities to instances, needed to serialize and unserialize references to instances that use the Java object identity as identifiers. Identities are unique among instances from the same modeling element (**EObject** subclass). Instances from a resource can reference instances from a different resource, provided that both resources belong to the same *Resource Set*. Each resource has an unique identifier, used as an index in the global *Registry*, another EMF concept introduced along with resources.

Since EMF does not distinguish models by their contents, i. e., another language syntax or real-world concepts, we refer to the contents of a model as *Instances* and to the modeling language elements as *Types*.

# 3 Model Consistency Approach

We consider a distributed system of interconnected nodes, where each node contains models expressed in different modeling languages. Nodes also contain resources, which are composed of instances from different modeling languages. In this system, any subset of nodes can share one or more resources: each node contains a replica of a shared resource and performs query and update operations on it. To ensure that all replicas of a shared resource are consistent, we propose the following approach:

1. Each update operation in a shared resource is first executed locally.
2. Thereafter, the operation is broadcast to all nodes containing replicas of the shared resource.
3. These nodes receive and integrate the update operation. The integration may result in undoing a locally executed operation, executing a new operation, and redoing that operation.

Shared resources are basically EMF resources that have replicas spread over a set of nodes and that are defined as follows:

**Definition 1** (Shared Resource)**.** *A* Shared Resource *is a tuple* $\mathcal{R} = \langle \textsc{rid}, N, I, F \rangle$ *where:* RID *is the resource unique identifier,* $N$ *is a set of nodes sharing the resource,* $I$ *is a set of instances, and* $F$ *is a set of features (i. e., instance attribute values and references between instances),*

Locally to each node, Java object identifiers (i. e., memory addresses) are commonly used as identifiers for instances and features. However, in a distributed environment, we must ensure the following propositions concerning the unique identification of resources, instances, types, and features.

**Proposition 1.** *Every node has an unique identity through the network, denoted by* NID*.*

The unicity of a NID may be ensured either locally, e. g., using a UUID, or distributively, e. g., using a naming server.

**Proposition 2.** *Every shared resource has an unique identity across the network, denoted by* RID*. This identity is independent from the node that created the resource and is ensured by a* Global Resource Registry*, which also helps nodes to find available shared resources.*

The registry is a simple associative array that may be implemented by a single node or by a Distributed Hash Table [23, 27].

**Proposition 3.** *Each instance has an unique identity across the network, denoted by* OID*, which depends on its containing resource and is independent from the node where it was created. An instance belonging to a shared resource has the same identity across all resource replicas.*

**Proposition 4.** *Each type and each feature of a type have unique identities across the network, denoted by* TID *and* FID*, respectively. A pair (*OID,FID*) identifies the value of feature* FID *on instance* OID.

The unicity of a type is usually ensured by its name and the name (or URI) of its modeling language. The unicity of a feature may be ensured by its name or by a natural number.

### 3.1 Update Operations

We consider only operations that modify the contents of a shared resource, i. e., operations that: add/remove instances to/from a resource, modify the values of instance monovalued features, or modify the valued of instance multivalued features. The specification of these operations is listed below:

- $\mathit{attach}(\text{RID}, \text{OID})$: adds instance OID to the shared resource RID.
- $\mathit{detach}(\text{RID}, \text{OID})$: removed an instance from a shared resource.
- $\mathit{set}(\text{OID}, \text{FID}, v)$: sets the value of feature FID to value $v$.
- $\mathit{unset}(\text{OID}, \text{FID})$: unsets feature FID.
- $\mathit{add}(\text{OID}, \text{FID}, v)$ adds value $v$ at the end of the multivalued feature FID.
- $\mathit{remove}(\text{OID}, \text{FID}, i)$: removes value of multivalued feature FID at index $i$.
- $\mathit{move}(\text{OID}, \text{FID}, s, t)$: moves value of multivalued feature FID from source index $s$ to target index $t$.
- $\mathit{clear}(\text{OID}, \text{FID})$: clears all values of multivalued feature FID.

Update operations can be formulated using simple mathematics. The following equation expresses the relation between a resource $\mathcal{R}$ and a resource $\mathcal{R}'$ that was modified by operation $O$.

$$\mathcal{R} = O * \mathcal{R}'$$

The operator "$*$" denotes the application of an update operation to a resource. Updating a resource means applying $n$ operations $O_i$ to a resource $\mathcal{R}'$ in a stepwise manner:

$$\mathcal{R} = O_1 * O_2 * \ldots * O_{n-1} * O_n * \mathcal{R}'$$

Two operations can be either dependent on, independent of, equivalent to or conflictual with each other. We define independent (or concurrent) operations as follows:

**Definition 2** (Independent Operations)**.** *Given any shared resource* $\mathcal{R}$ *and any two operations* $O_a$ *and* $O_b$ *are said to be* independent *of each other if they are commutative, i. e., if an only if* $O_a * O_b * \mathcal{R} = O_b * O_a * \mathcal{R}$.

Conceptually, each operation $O$ is associated to an original context $C_O$, i. e., the sequence of operations required to bring a resource from its initial state to the state where $O$ can be applied.

**Definition 3** (Dependent Operations)**.** *Given any operations* $O_a$ *and* $O_b$*, and* $C_{O_a}$*, the original context of operation* $O_a$*,* $O_a$ *is said to be* dependent *on* $O_b$ *if and only if* $O_b \in C_{O_a}$.

When two operations have the same original context and are not independent, they are said to be conflictual. For instance, operations $set(\text{OID}_a, \text{FID}_1, v_a)$ and $set(\text{OID}_a, \text{FID}_1, v_b)$ are conflictual.

**Definition 4** (Conflictual Operations)**.** *Given any shared resource* $\mathcal{R}$ *and any two operations* $O_a$ *and* $O_b$ *and their original contexts* $C_{O_a}$ *and* $C_{O_b}$*,* $O_a$ *and* $O_b$ *are said to be* conflictual *if and only if* $C_{O_a} = C_{O_b}$ *and* $O_a * O_b * \mathcal{R} \neq O_b * O_a * \mathcal{R}$

In most cases, operations have different contexts and therefore are independent. For instance, the operations *set* and *remove* both concern features, but since features cannot be mono and multivalued at the same time, they are obligatory independent.

Some operations may produce the same result, even when they come from different nodes. For instance, two operations *clear*, or two operations *remove* or *add* of the same value, produce the same results on the same features.

**Definition 5** (Equivalent Operations)**.** *Given any shared resource* $\mathcal{R}$ *and any two operations* $O_a$ *and* $O_b$ *and their original contexts* $C_{O_a}$ *and* $C_{O_b}$*,* $O_a$ *and* $O_b$ *are said to be* equivalent *if and only if* $C_{O_a} = C_{O_b}$ *and* $O_a * \mathcal{R} = O_b * \mathcal{R}$

## 3.2 Casual Dependencies

The casual dependency relation, denoted by "$\rightarrow$", expresses that one operation happened before another and is commonly based on time [29, 17]. In our approach, we adopt a semantic casual dependency [16, 20]. The idea is not to establish whether a given operation $O_a$ at node $n_1$ was generated before operation $O_b$ at node $n_2$, but whether $O_b$ depends on $O_a$. For instance, the operation $O_a = attach(\text{RID}_1, \text{OID}_a)$ precedes operation $O_b = set(\text{OID}_a, \text{FID}_1, value)$, $O_a \rightarrow O_b$, since object $\text{OID}_a$ must exist before feature FID is set. Conversely, two operations $O_a$ and $O_b$ are said to be independent (or concurrent), if and only if neither $O_a \rightarrow O_b$ nor $O_b \rightarrow O_a$, which is expressed as $O_a \parallel O_b$.

In our approach, we adopt following propositions concerning the semantic casual dependencies between conflictual operations. In these propositions, we assume that the operations have the same original contexts.

Let us denote by $O^{Attach(i)}$ an operation that attaches an instance $i$ to a resource, by $O^{Detach(i)}$ an operation that detaches an instance $i$ from a resource, and by $O^{Any}$ any feature-related operation.

**Proposition 5.** *For any Instance i, we have the following semantic casual dependency:* $O^{Attach(i)} \rightarrow O^{Any(i)} \rightarrow O^{Detach(i)}$

Two *attach*() operations cannot be conflictual, since instances attached to different shared resource replicas have different identifications, according to Proposition 3. Two *detach*() operations are equivalent since they produce the same result.

There is no semantic casual dependency between Operations on monovalued features with the same original context, *set* and *unset*. However, it can be established for operations on multivalued features, *add*, *remove*, *clear*, and *move*.

Let us denote by FID a multivalued feature, by $O^{Add(\text{FID})}$ an operation and adds a value to FID, by $O^{Remove(\text{FID})}$ an operation that removes an element from FID, by $O^{Clear(\text{FID})}$ an operation that clears FID, and by $O^{Move(\text{FID})}$ an operation that moves around a value in FID.

**Proposition 6.** *For any multivalued feature* FID*, we have the following casual dependencies:*

- $O^{Move(\text{FID})} \rightarrow O^{Remove(\text{FID})}, O^{Clear(\text{FID})}$
- $O^{Move(\text{FID})} \parallel O^{Add(\text{FID})}$
- $O^{Clear(\text{FID})} \rightarrow O^{Add(fid)}$
- $O^{Remove(\text{FID})} \rightarrow O^{Clear(\text{FID})}$
- $O^{Add(\text{FID})} \parallel O^{Remove(\text{FID})}$

Differently from the other operations, $O^{Move}$ parameters are indexes, instead of values. Therefore, any operation that changes the position of a value affects the behavior of $O^{Move}$. In the opposite, $O^{Move}$ operations do not affect operations that use values as parameters. $O^{Move}$ and $O^{Add}$ are independent, since a value is added to the end of the feature and do not affect a move operation. $O^{Clear(fid)}$ precedes $O^{Add(fid)}$ because when the first operation is executed, it is not aware of the value added by the second one. $O^{Remove(\text{FID})}$ precedes $O^{Clear(\text{FID})}$, otherwise the first operation could raise an error (value not found). Finally, $O^{Add(\text{FID})}$ and $O^{Remove(\text{FID})}$ are independent, even if their arguments are the same. Indeed, the first operation adds a value to the end of a feature, while the second one removes the first occurrence of a value.

In complement to the casual dependency between operations from different types, we have the following casual dependencies between operations of the same type:

- Two *add* or two *remove* operations are either independents or equivalents.
- Two *clear* operations are equivalents.
- Two *move* operations are independents if the range of values between the source and the target indices do not overlap.

## 3.3 Integration Algorithm

To propagate local changes to remote nodes, nodes send an *update messages* for each operation executed locally. We define update messages as follows.

**Definition 6.** *An* Update Message *is a tuple* $\mathcal{M} = \langle n, \mathcal{R}, O, C \rangle$ *where:* $n$ *is the source node,* $\mathcal{R}$ *the shared resource,* $O$ *is the executed operation, and* $C_O$ *is the operation original context.*

The integration requires that each node implements a precedence relation, according to the following proposition:

**Proposition 7.** *For all nodes sharing a resource, there is a precedence relation denoted by "*$\prec$*",* $\prec: \mathcal{M} \times \mathcal{M} \rightarrow \mathbb{B}$*, such as for any pair of update messages* ($m_a$,$m_b$)*,* $m_a \prec m_b$ *produces the same result in all nodes.*

A simple way to ensure that the precedence operator behaves the same in all nodes is to use properties belonging to the message: e. g., the source node, the operation arguments, a hash function on the arguments, etc.

The integration also requires that each node implements a context-equivalent relation, according to the following proposition:

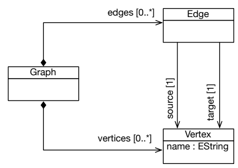
**Proposition 8.** *For all nodes sharing a resource, there is a context-equivalent relation denoted by "*$\sqcup$*",* $\sqcup: \mathcal{M} \times \mathcal{M} \rightarrow \mathbb{B}$*, such as for any pair of update messages* ($m_a$,$m_b$)*,* $m_a \sqcup m_b$ *if and only if* $C_{m_a} = C_{m_b}$*.*

Algorithm 1 describes the integration of update messages on nodes. Each node has a local history of integrated remote messages, denoted by $\mathcal{H}$ and receives an update message $m$. The integration first verifies if an equivalent message exists in $\mathcal{H}$ and stops the integration if it is the case. Then, it searches all messages that are context-equivalent with $m$ and that should precede $m$, adds these messages to the set *successors* and removes them from $\mathcal{H}$. After the removal, message $m$ is executed and added to $\mathcal{H}$. Lastly, the integration re-executes all *successors* and adds them to $\mathcal{H}$.

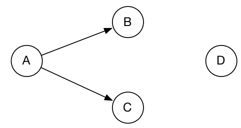
**Algorithm 1:** Update Message Integration

```
Input: m, an Update Message; H, the local history.
if ∃h, h ∈ H ∧ h ≡ m then
    return
successors ← {h | h ∈ H ∧ m ≺ h ∧ m ⊔ h};
H ← H − successors ;
foreach each ∈ successors do
    undo(each)
execute(m);
foreach each ∈ successors do
    execute(each)
H ← H + {m} + successors;
```

### 3.4 Example




(a) GraphML Syntax in Ecore




(b) A GraphML Instance

Figure 1: Simple Example

Figures 1a and 1b present respectively the Ecore model for a Graph modeling language (GraphML) and a model containing an instance of this language, i. e., a graph. This graph contains 7 instances, each one with an unique identifier:

- the graph itself, identified by $g$.
- 4 vertices (and their identifiers): "A" ($a$), "B" ($b$), "C" ($c$), and "D" ($d$).
- 2 edges, identified by $ab$ and $ac$.

Let us suppose a collaborative environment, where a shared resource containing this graph is being modified by three different nodes, performing the following modifications:

**Node 1** : renames vertex $a$ to "A1".

**Node 2** : renames vertex $a$ to "A2" and deletes vertex $d$.

**Node 3** : creates a new vertex $e$, named "E", and adds it to graph $g$; creates a new edge $ae$ between $a$ and $e$ and adds it to graph $g$; and deletes vertex $d$.

Table 1 presents a summary of the operations generated by these modifications. These operations are first executed locally at each node and then broadcast to the other nodes. We present the integration of operations on each node in the next sections. In this example, the order nodes receive remote operations from remote nodes is arbitrary. Nevertheless, if different orders occurs, the integration result would be the same.

### 3.5 Integration at Node 1

Node 1 receives operations $O^2_{1..3}$ from Node 2 and integrates them sequentially. Operations $O^1_1$ and $O^2_1$ conflict: they both modify the value of the same feature and have equivalent contexts. Node 1 uses the precedence relation to determine that $O^1_1 \prec O^2_1$ and executes operation $O^2_1$. Operations $O^2_2$ and $O^2_3$ are not conflictual with the precedent ones and are executed.

| **Node 1** | **Node 2** | **Node 3** |
|---|---|---|
| $O_1^1 = set(a, \#name, "A1")$ | $O_1^2 = set(a, \#name, "A2")$<br>$O_2^2 = remove(g, \#vertices, d)$<br>$O_3^2 = detach(d)$ | $O_1^3 = attach(e)$<br>$O_2^3 = set(e, \#name, "E")$<br>$O_3^3 = add(g, \#vertices, e)$<br>$O_4^3 = attach(ad)$<br>$O_5^3 = add(g, \#edges, ae)$<br>$O_6^3 = set(ad, \#target, e)$<br>$O_7^3 = set(ad, \#source, a)$<br>$O_8^3 = remove(g, \#vertices, d)$<br>$O_9^3 = detach(d)$ |

Table 1: Summary of Operations at Nodes 1, 2, and 3.

Then, Node 1 receives operations $O^3_{1..9}$ from Node 3. Operations $O^3_1$ and $O^3_2$ concern a new instance, are independent and are executed. $O^3_3$ and $O^2_2$ concern the same feature from the same instance, however, they are independent (Proposition 6) and $O^3_3$ is executed. $O^3_{4..7}$ are all independent and are executed. Operation $O^3_8$ is equivalent to $O^2_2$ and $O^3_9$ is equivalent to $O^2_3$. Both operations are not executed. This results in the following history of operations:

$$\mathcal{H}_1 = \left\{O^1_1, O^2_1, O^2_2, O^2_3, O^3_1, O^3_2, O^3_3, O^3_4, O^3_5, O^3_6, O^3_7\right\}.$$

### 3.6 Integration at Node 2

Node 2 receives operations $O^3_{1..9}$ from Node 3. Similarly to the precedent integration at Node 1, Node 2 executes operations $O^3_{1..7}$, which are independent and does not execute operations $O^3_8$ and $O^3_9$, which are equivalent to $O^2_2$ and $O^2_3$.

Then, Node 2 receives $O^1_1$ from Node 1, which conflicts with operation $O^2_1$. Node 2 uses the same precedence relation as Node 1 to determine that $O^1_1 \prec O^2_1$ and cannot execute operation $O^1_1$. It first undoes operation $O^2_1$, executes $O^1_1$ and re-executes $O^2_1$. This results in the following history of operations:

$$\mathcal{H}_2 = \left\{O^2_2, O^2_3, O^3_1, O^3_2, O^3_3, O^3_4, O^3_5, O^3_6, O^3_7, O^1_1, O^2_1\right\}.$$

### 3.7 Integration at Node 3

Lastly, Node 3 receives and integrates operations $O^2_{1..3}$ from Node 2, without executing $O^2_2$ and $O^2_3$. Then, it receives $O^1_1$ from Node 1, which conflicts with operation $O^2_1$, as in the other nodes. The very same precedent relation determines that $O^1_1 \prec O^2_1$ and operation $O^1_1$ cannot be executed. Thus, Node 3 first undoes operation $O^2_1$, and then executes $O^1_1$ and re-executes $O^2_1$, resulting in the following history of operations:

$$\mathcal{H}_3 = \left\{O^3_1, O^3_2, O^3_3, O^3_4, O^3_5, O^3_6, O^3_7, O^3_8, O^3_9, O^1_1, O^2_1\right\}.$$

### 3.8 Discussion

After integration, all three nodes have equivalent replicas of the same shared resources, all three local histories are equivalent ($\mathcal{H}_1 \equiv \mathcal{H}_2 \equiv \mathcal{H}_3$), ensuring convergence and intention preservation. The integration algorithm ensures that in all nodes, the only pair of conflictual operations, $(O_1^1, O_1^2)$, is executed in the same sequence, i. e., in all nodes $O_1^1 \rightarrow O_1^2$.

If Node 1 is not satisfied with the name of Vertex $a$ and renames it again, creating operation $O_2^1 = set(a, \#name, "A1")$, this operation is broadcast and executed on the other nodes without conflicts. Indeed, since both operations $(O_1^1, O_1^2)$ belong to the original context of $O_2^1$, i. e., $O_2^1$ depends on $O_1^1$ and on $O_1^2$ (Definition 3).

# 4 Prototype Implementation

To validate the integration algorithm, we develop a prototype in Java (v. 1.8), based on EMF (v. 2.12). While the algorithm could be implemented in other languages and other modeling frameworks, we choose EMF to benefit from resource management and the change notification framework. We use the distributed hash table TomP2P DHT [5] to implement the distributed shared resource registry and the HornetQ messaging system [11] to broadcast change messages. The initial validation of the prototype uses PeerUnit [1], a distributed test architecture.

In this section, we present the main design and implementation choices adopted for the prototype. The source code is available on GitHub[1].

### 4.1 Identities

In EMF, types and features are identified by integer numbers, associated to a package (**EPackage**). A package is a *Façade* [10] for the generated underlying model. It uses a namespace URI, originated from the source Ecore model, as identity. Thus, types (and features) can be identified by a URI and one (or two) integers. Similarly to packages, instances also use a URI as an identity, when no indentity attribute exists.

While using URI as identities ensures their unicity, URI are long strings which are not adapted for network message exchanges. To avoid this problem and use more efficient identities, we introduce a distributed version of the package registry. This class is basically a map that allows retrieving packages from its Id and an Id from the package URI. The shared resource class is also a map that allows retrieving instances from their Id. Figure 2 sketches these two classes.

To ensure the unicity of an instance Id, we adopt the high-low strategy [2]. The identity of an instance is then the Id of the shared resource it is attached to (high part) and an unique identifier within this resource (low part). This same

[1] `https://github.com/sunye/model-consistency`

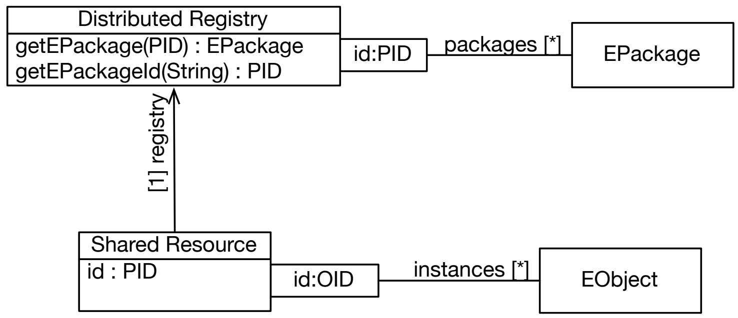



Figure 2: Distributed Registry

strategy is used for types and features. Figure 3 sketches these identities and their relationships.

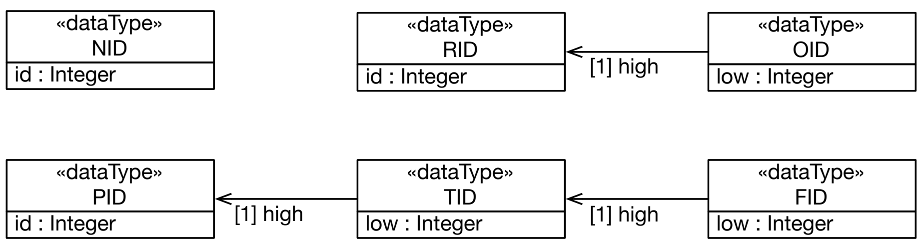



Figure 3: Identity Datatypes

We use EMF adapters to associate an OID to instances when they are attached to a shared resource, avoiding the modification of the different **EObject** implementations.

## 4.2 Update Notification

The EMF change notification framework is an enhanced implementation of the Observer and the Adapter design patterns [10], where the adapter class is also an observer. When any feature of an instance is changed, its adapters receive informations about the change.

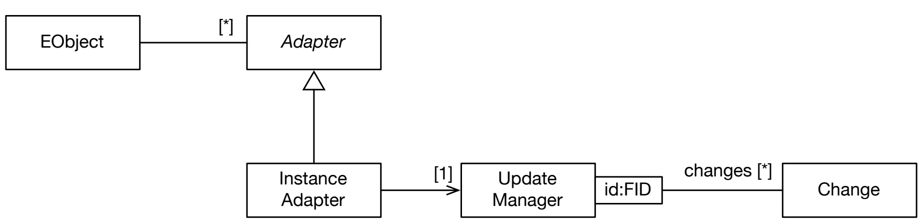



Figure 4: Update Notification

Figure 4 depicts a UML class diagram representing the update notification mechanism. When an instance is changed, the instance adapter receives a notification and forwards it to the update manager. The latter stores the change, which is later broadcast to remote nodes through the Publish-subscribe service.

### 4.3 Original Context and Precedence

To detect conflicts between operations, each operation is associated to an original context, i. e., the state of the shared resource when the change was done. We adopt two different strategies to establish the original context, both based on the changed feature. For operations on manovalued features, we use the previous value of the feature. According to this rationale, two operations have the same original context if the previous values of the concerned feature are the same.

Multivalued features are more complex, since sending all values of a collection would be too expensive. In this case, hashing the collection values is a more efficient alternative, albeit still expensive. We adopt an alternative strategy, which consists in keeping track of the node that originated the last change and of the of the number of times the feature has been structurally modified (analogously to the **modCount** field in the Java **AbstractList** class).

To determine the precedence relation between two conflictual operations, we adopt a straightforward strategy, we use a hash function to calculate the hash values of the operation values. The operation that has the lower hash value precedes the operation with the greater one. This strategy works for operations on mono and multivalued features, except for the *move* operation, which does not have any associated value. In this case, we first compare the source indices and if they are equal, we compare the target indices. An operation with the lower index precedes the one with the greater index.

## 5 Related Work

Standard control version systems, e. g., CVS, Subversion, or Git, are not fully adapted for collaborative modeling. Although models can be exported to XMI, a textual format that could be managed by a version system, this approach would not be successful. Indeed, XMI files are generated dynamically and this generation does not ensure neither that the order of XML tags nor that tag identification attributes remain unchanged across different generations. In consequence, the version system may detect several conflicts on two XMI files representing the same model.

To avoid these issues, academic and industrial projects developed control version systems dedicated to models. EMFStore [12] from TU Munich, ModelCVS [14] from TU Vienna, MetaEdit+ [31], and Modelio Constellation [8] from Softeam implement RCS' well-oiled checkout-update-commit pattern for EMF resources. They consider the semantics of modeling languages and thus can correctly support model merging and conflict detection. They differ from our tool by supporting asynchronous cooperative work, while we focus on synchronous cooperative work.

The EMFStore project also proposes a synchronous real-time extension [15], based on the Bonjour peer-to-peer protocol. While their project has the same goal as ours, they adopt a different approach for change integration on nodes, which is based on Git. More precisely, they use hash values to identify change operations (packages) and maintain a reference to the parent operation. When conflicts occur, the tool asks the user to solve them. We believe that our semantic casual dependency is more pertinent for detecting conflicts and that the use of local context information instead of hash numbers consumes less resources.

Koshima et al. propose DiCoMEF [13], a collaborative model-editing framework. Similarly to our approach, this tools detects conflicts at a low granularity level, the update operations. Unlike our approach, operations can be annotated with multimedia information to help users to manually solve conflicts.

Model repositories such a Morsa [21] and Eclipse CDO [25] use a pessimistic locking approach as a support for collaborative modeling. In this centralized approach, users lock the elements they want to edit, preventing others from accessing these elements. Chechik et al. propose the use of a property locking approach for more efficient locking [6]. They use the semantic of the modeling language to avoid users to introduce changes that could generate inconsistencies for other users.

Hawk [3] is a distributed model indexing framework for file-based models. Hawk uses a NoSQL database to store and update continuously metadata information from these models, to provide efficient and scalable model querying.

# 6 Conclusion and Future Work

The model consistency approach presented in this paper is an initial step towards effective collaborative modeling. However, a large amount of work still remains. Currently, the approach does not ensure the security of the system and does not provide a service to send efficiently large resources through the network. This is an issue when nodes open shared resources with an important initial size.

Additionally, the approach does not consider some syntax rules that are specific to modeling languages. For instance, if two software modelers are editing the same UML diagram and create two classes with same name, this would not be considered as an error, since these classes would have different identities. However, the diagram would not be valid according to the UML wellformed rules.

The approach adopts a data consistency algorithm, where changes are small and conflicts are automatically solved. The approach must be integrated into existing modeling tools to evaluate the impact of these choices on the usability of the tools during collaborative modeling. Furthermore, we want to analyze the impact of these choices when performing a complex sequence of changes, e. g., when performing different refactorings on UML models [30].

As future work, we will integrate the approach to NeoEMF [7] and extend it to provide a distributed repository of models, as well as a service to allow inter-resource references.

# References


[1] de Almeida, E.C., Sunyé, G., Le Traon, Y., Valduriez, P.: Testing peer-to-peer systems. Empirical Software Engineering 15(4), 346–379 (2010)

[2] Ambler, S.W.: The object primer: Agile model-driven development with UML 2.0. Cambridge University Press, 3rd edition edn. (2004)

[3] Barmpis, K., Kolovos, D.S.: Towards scalable querying of large-scale models. In: European Conference on Modelling Foundations and Applications. pp. 35–50. Springer (2014)

[4] Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. SIGOPS Oper. Syst. Rev. 21(5), 123–138 (Nov 1987), `http://doi.acm.org/10.1145/37499.37515`

[5] Bocek, T.: Tomp2p a p2p-based high performance key-value pair storage library (February 2017), `https://tomp2p.net/`

[6] Chechik, M., Dalpiaz, F., Debreceni, C., Horkoff, J., Ráth, I., Salay, R., Varró, D.: Property-based methods for collaborative model development. In: Joint Proceedings of the 3rd International Workshop on the Globalization Of Modeling Languages and the 9th International Workshop on Multi-Paradigm Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, GEMOC+MPM@MoDELS 2015, Ottawa, Canada, September 28, 2015. pp. 1–7 (2015), `http://ceur-ws.org/Vol-1511/paper-01.pdf`

[7] Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: Neoemf: a multi-database model persistence framework for very large models. In: Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016. pp. 1–7 (2016), `http://ceur-ws.org/Vol-1725/demo1.pdf`

[8] Desfray, P.: Model repositories at the enterprises and systems scale: The modelio constellation solution. In: 2015 International Conference on Information Systems Security and Privacy (ICISSP). pp. IS–17–IS–17 (Feb 2015)

[9] Dirix, M., Muller, A., Aranega, V.: GenMyModel : An Online UML Case Tool. ECOOP (2013), `https://hal.archives-ouvertes.fr/hal-01251417`, poster

[10] Gamma, E., Helm, R., Johnson, R., Vlissides", J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional (1995)

[11] Giacomelli, P.: Hornetq messaging developer's guide. Packt Publishing Ltd (2012)

[12] Koegel, M., Helming, J.: Emfstore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. pp. 307–308 (2010), `http://doi.acm.org/10.1145/1810295.1810364`

[13] Koshima, A.A., Englebert, V.: Collaborative editing of emf/ecore metamodels and models: Conflict detection, reconciliation, and merging in dicomef. Sci. Comput. Program. 113, 3–28 (2015), `http://dx.doi.org/10.1016/j.scico.2015.07.004`

[14] Kramler, G., Kappel, G., Reiter, T., Kapsammer, E., Retschitzegger, W., Schwinger, W.: Towards a semantic infrastructure supporting model-based tool integration. In: Proceedings of the 2006 International Workshop on Global Integrated Model Management. pp. 43–46. GaMMa '06, ACM, New York, NY, USA (2006), `http://doi.acm.org/10.1145/1138304.1138314`

[15] Krusche, S., Brügge, B.: Model-based real-time synchronization. Softwaretechnik-Trends 34(2) (2014), `http://pi.informatik.uni-siegen.de/stt/34_2/01_Fachgruppenberichte/CVSM2014/KruscheCVSM2014.pdf`

[16] Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. 10(4), 360–391 (1992), `http://doi.acm.org/10.1145/138873.138877`

[17] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978), `http://doi.acm.org/10.1145/359545.359563`

[18] Li, R., Li, D.: Commutativity-based concurrency control in groupware. In: Zhang, T. (ed.) Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing, San Jose, CA, USA, December 19-21, 2005. IEEE Computer Society / ICST (2005), `http://dx.doi.org/10.1109/COLCOM.2005.1651251`

[19] Matula, M.: Netbeans metadata repository. Tech. rep., Sun Microsystems (2003)

[20] Oster, G., Urso, P., Molli, P., Imine, A.: Data consistency for P2P collaborative editing. In: Hinds, P.J., Martin, D. (eds.) Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006,

Banff, Alberta, Canada, November 4-8, 2006. pp. 259–268. ACM (2006), http://doi.acm.org/10.1145/1180875.1180916

[21] Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. In: Proceedings of the 14th MoDELS Conference. pp. 77–92. Wellington, New Zealand (2011), http://dl.acm.org/citation.cfm?id=2050655.2050665

[22] Plotnikov, C.: Novosoft metadata framework and uml library (2002), http://nsuml.sourceforge.net

[23] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 161–172. ACM, New York, NY, USA (2001)

[24] Software, L.: (March 2017), https://www.lucidchart.com

[25] Steinberg, D.: Fundamentals of the eclipse modeling framework. Tutorial presented at EclipseCon 2008 (March 2008), http://www.eclipsecon.org/2008/index1000.html

[26] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF – Eclipse Modeling Framework. The Eclipse series, Pearson Education, 2nd edn. (2008)

[27] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 149–160. ACM, New York, NY, USA (2001)

[28] Sun, C., Ellis, C.A.: Operational transformation in real-time group editors: Issues, algorithms, and achievements. In: CSCW '98, Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work, Seattle, WA, USA, November 14-18, 1998. pp. 59–68 (1998), http://doi.acm.org/10.1145/289444.289469

[29] Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput.-Hum. Interact. 5(1), 63–108 (Mar 1998), http://doi.acm.org/10.1145/274444.274447

[30] Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.: Refactoring UML models. In: UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings. pp. 134–148 (2001), http://dx.doi.org/10.1007/3-540-45441-1_11

[31] Tolvanen, J.P.: Metaedit+ for collaborative language engineering and language use (tool demo). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. pp. 41–45. ACM (2016)